Specifying time-sensitive systems with TLA+

Hehua Zhang, Ming Gu, Xiaoyu Song

2010 34th Annual IEEE Computer Software and Applications Conference

Specifying time-sensitive systems with TLA+

Hehua Zhang
School of Software, KLISS, TNLIST
Tsinghua University & INRIA
Beijing, China & Nancy, France
Email: zhanghehua@gmail.com

Ming Gu
School of Software, KLISS, TNLIST
Tsinghua University
Beijing, China
Email: guming@tsinghua.edu.cn

Xiaoyu Song
Dept. ECE
Portland State University
Oregon, USA
Email: song@ee.pdx.edu

***Abstract*—We present a pattern-based method to express time specifications in the language TLA+. A real-time module *RealTimeNew* is introduced to encapsulate the definitions of commonly used time patterns. We present a general framework to differentiate the temporal characterizations from system functionality with time constraints. The temporal specification is concise and provably as a refinement of its corresponding functional description without time. The method ameliorates the usability of TLA+ in specifying and verifying time-sensitive systems. A case study is harnessed to illustrate and validate the approach.**

***Keywords*-real-time; specification; TLA+; refinement**

I. Introduction

TLA+ [1] is a formal specification language, which is based on the Temporal Logic of Actions TLA [2], first-order logic, and Zermelo-Fränkel set theory. It was designed for specifying and reasoning about concurrent and reactive systems and was widely used in many fields [3], [4].

With the wide use of real-time systems, embedded systems and pervasive computing in everyday applications, time-sensitive systems (i.e. systems whose behavior is influenced by the passing of time) attract many people's attention. To formally analyze time-sensitive systems, it is necessary to represent time in the formalisms. Carlo [5] presents a comprehensive survey on the time modeling methods in computing. Among various approaches, the TLA+ language shows both operational and descriptive features [6] in a single logic framework. It is thus suitable to describe both the evolution of a system and the properties to be satisfied.

Abadi and Lamport introduced firstly the time modeling format in the TLA logic [7]. Different with many implicit-time models like timed automata [8] and time Petri nets [9], they explicitly represents the real time with a state variable *now*, and describes the expected behaviors with respect to the occurrence of the actions and the current time *now*. This format was then taken to specify several kinds of time-sensitive systems [10], [11], but the time specification method is still ad hoc. Lamport developed a real-time module *RealTime* [12] to ease the time modeling applications. However, it can be used to specify only the time duration of an action, not the time intervals between actions, which are also common for time-sensitive systems.

In this paper, we present a method to ameliorate the usability of TLA+ in specifying and verifying time-sensitive systems[1]. A real-time module *RealTimeNew* is introduced to encapsulate common time patterns. The basic patterns specify the time duration of an action or the time interval between actions. Advanced time patterns are further defined based on the basic ones. We then present a general framework to differentiate the temporal characterizations from system functionality with time constraints. The temporal specification is concise and provably as a refinement of its corresponding functional description without time. Furthermore, the obtained time specifications can be checked directly by the TLC model checker, when limited to a finite domain.

The paper is organized as follows. The preliminaries of TLA+ are introduced in Section II. We define the real-time module *RealTimeNew* in Section III. The framework of time specification is presented in Section IV. Section V presents a case study that validates our approach and introduces the verification. Finally, we conclude the work in Section VI.

II. Preliminaries of TLA+

The characteristic form of the TLA+ specification of a transition system is a formula of the form

$$Spec \triangleq Init \wedge \Box[Next]_{vars} \wedge L,$$

where $vars$ is a tuple containing all state variables of the system. The first conjunct $Init$ describes the possible initial states of the system. The second conjunct of the specification asserts that every step (i.e., every pair of successive states in a system run) either satisfies $Next$ or leaves the term $vars$ (and therefore all state variables) unchanged. Allowing for such stuttering steps is a key ingredient to obtain compositionality of specifications. However, it means that executions that stutter indefinitely are allowed by the specification. The third conjunct L is a temporal formula stating the liveness conditions of the specification, and in particular can be used to rule out infinite stuttering.

[1]This research is sponsored in part by NSFC Program (No.90718039) and 973 Program (No.2010CB328003) of China.

0730-3157/10 $26.00 © 2010 IEEE
DOI 10.1109/COMPSAC.2010.50

III. MODELING TIME IN TLA$^+$

A real-time module *RealTimeNew* is provided to model time in TLA$^+$. The format of it is inspired by Lamport's *RealTime* module, but there is a basic difference. The *RealTime* module is designed to be used through specification composition [12], while we design the *RealTimeNew* module to be used in a single specification, so that it can be verified directly by the TLC model checker (on a finite domain) or deductive verification. *RealTimeNew* also encapsulates richer time patterns and can then be applied to more applications.

The time formulas are classified into four kinds: time evolving, the time duration of an action, the time interval between actions and advanced time patterns.

A. Time evolving

The time evolving representation follows the one in the *RealTime* module [12], with the real time is modeled by a real-valued variable *now*:

$$NowNext(v) \triangleq \land\ now' \in \{r \in Real : r > now\} \\ \land\ \text{UNCHANGED}\ v.$$

The action *NowNext(v)* changes the value of now by any real number r satisfying $r > now$, while leaves the parameter v unchanged. The parameter v denotes a single variable or a tuple of variables. Furthermore, Zeno behaviors [7] of time need to be ruled out, since the real time increases unboundedly. To ensure non-Zenoness, a fairness constraint is defined as

$$RTFairness(v) \triangleq \\ \forall r \in Real : WF\_now(NowNext(v) \land (now' > r)).$$

$WF_v(A)$ represents the weak fairness constraint, which means that if the action A is continuously enabled, then it will eventually happen. Since the action $(NowNext(v) \land (now' > r))$ is always enabled, *RTFairness(v)* denotes that a *NowNext(v)* action will eventually occur, ensuring the non-Zenoness.

B. Time duration of an action

Constraints on the time duration of an action is common in time-sensitive systems. For example, "The duration of an hour step of an electric clock should be between 59.9 minutes to 60.1 minutes". To represent this kind of time constraints, four time patterns are defined, respectively.

A general time duration constraint of an action A is a range (with both lower bound and upper bound). For formal specifying this time constraint, we face the question that whether the action A is forced to occur, when the upper bound is reached with time elapsing ? Following the convention, we call it the *strong time request (STR)* when the answer is "Yes" while the *weak time request (WTR)* otherwise. Since *STR* and *WTR* are both possible in applications, boht of them are considered in the definition of the time range constraint *DurationBound*:

$$\begin{array}{l} DurationBound(STRflag, t, A, v, min, lb, max, ub) \triangleq \\ \text{LET} \\ \quad TNext \triangleq t' = \text{IF}\ \langle A\rangle_v \lor \neg(\text{ENABLED}\ \langle A\rangle_v)' \\ \qquad\qquad\qquad \text{THEN}\ 0 \\ \qquad\qquad\qquad \text{ELSE}\ t + (now' - now) \\ \quad UpperBound \triangleq \\ \qquad \text{IF}\ STRflag = \text{TRUE} \\ \qquad\ \text{THEN IF}\ ub\ \text{THEN}\ t' \leq max\ \text{ELSE}\ t' < max \\ \qquad\ \text{ELSE}\ A \Rightarrow \text{IF}\ ub\ \text{THEN}\ t \leq max\ \text{ELSE}\ t < max \\ \quad LowerBound \triangleq \\ \qquad A \Rightarrow \text{IF}\ lb\ \text{THEN}\ t \geq min\ \text{ELSE}\ t > min \\ \text{IN} \quad \land\ TNext \\ \qquad\ \land\ UpperBound \\ \qquad\ \land\ LowerBound \end{array}$$

The definition takes eight parameters. The first parameter is a *STR* flag, with the value TRUE denoting *STR* and the value FALSE otherwise. A denotes the action to be considered. The time duration of A is recorded by a timer t, and is requested to be between the range $[min, max]$. The parameters lb and ub are boolean flags denoting the boundary value min and max is included or not, respectively. The parameter v is a single variable or a tuple of variables.

The *DurationBound* action is composed by the conjunction of three sub-formulas: *TNext*, *UpperBound* and *LowerBound*. *TNext* denotes how the timer t is changed. According to its definition, the new value of t is reset to zero when A occurs or will be disabled in the next step. Otherwise, A is continuously enabled, so that the timer t accumulates its value with time elapsing. $\langle A\rangle_v$ equals to $A \land v' \neq v$, denoting the actual execution of A, excluding stuttering steps.

UpperBound denotes the upper bound of the time duration. When it is a *STR*, *UpperBound* specifies that the next value of t must be always less than (or equal to) max, and thus ensures the happending of A when the upper bound is reached. When it is a *WTR*, the formula only specifies that the occurrence of A implies $t \leq max$. As a result, A may happen but not requested when the upper bound is hit.

LowerBound requests that if A happens, the current value of t must be greater than (or equal to) min, and thus expresses the lower bound on the duration. In sum, the conjunction of *TNext*, *UpperBound* and *LowerBound* defines a range constraint on the time duration of an action A.

For example, "The duration of an hour step of an electric clock should be between 59.9 minutes to 60.1 minutes" can be represented by an instantiated action: *DurationBound(*TRUE*, t, Step, v, 59.9,* FALSE*,60.1,* FALSE*)*, where $Step$ denotes the hour stepping action.

The other three actions are variations of *DurationBound*, for ease of use. The formula *DurationUB* specifies the case when only the upper bound constraint of a time

duration is required. Dually, we provide the formula $DurationLB(t, A, v, min, lb)$ to describe the case when only the lower bound of a time duration is required. The fourth action *DuratonValue* is used to represent the exact value request on a time duration.

C. Time interval between two actions

Constraints on the time interval between two actions are also common for time-sensitive systems. For example, the period of time between the occurrence of the action B and the action D should be always greater than 2.5 seconds; The actions C happens within 3 seconds of the preceding action A. The *RealTimeNew* module also contains four actions for the different possible time constraints on time intervals.

The fist formula *IntervalBound* specifies a range of the time interval, and both *STR* and *WTR* are considered. When considering the time interval between A and B with the precedence relationship, *STR* requests that when the upper bound is reached, B forced to occur, while *WTR* not. Another question is: when we talking about the time interval between the actions A and B, is the time computed by the beginning of the execution or the end of the execution of A (or B)? There are several possibilities. The time interval semantics we take in this paper is from the end of A to the end of B. As a result, the action *IntervalBound* specifying a range of a time interval can then be defined as follows.

$$
\begin{array}{l}
IntervalBound(STRflag, t, Acts, B, v, min, lb, max, ub) \triangleq \\
\text{LET } TNext \triangleq t' = \text{IF } \langle B \rangle_v \vee \neg(\text{ENABLED } \langle Acts \vee B \rangle_v)' \\
\qquad\qquad\qquad\qquad \text{THEN } 0 \\
\qquad\qquad\qquad\qquad \text{ELSE } t + (now' - now) \\
\quad UpperBound \triangleq \\
\qquad \text{IF } STRflag = \text{TRUE} \\
\qquad\ \text{THEN IF } ub \text{ THEN } t' \leq max \text{ ELSE } t' < max \\
\qquad\ \text{ELSE } B \Rightarrow \text{IF } ub \text{ THEN } t \leq max \text{ ELSE } t < max \\
\quad LowerBound \triangleq \\
\qquad B \Rightarrow \text{IF } lb \text{ THEN } t \geq min \text{ ELSE } t > min \\
\text{IN} \quad \wedge\ TNext \\
\qquad \wedge\ UpperBound \\
\qquad \wedge\ LowerBound
\end{array}
$$

To compute the total time from the ending of A to the ending of B, it is required to know the actions possibly happen between A and B. Suppose there are n actions possibly happen between A and B, denoted by $A_1, \ldots, A_n$, respectively , we can describe the time interval between the action A and B by taking $Acts \triangleq A_1 \vee \ldots, \vee A_n$. The time interval is recorded by the timer t.

According to the definition of *TNext*, the new value of t is reset to zero when B happens or $Act \vee B$ will be disabled in the next step. Otherwise, the timer t accumulates its value with time elapsing. As to all the actions that neither makes an action between A and B happens nor makes B happens, the timer t does not count. As a result, only the elapsing time related to the interval is counted. The meaning of other parameters are same to the ones in $DurationBound$.

For example, considering the four actions A, B, C and D which happen sequentially, the request that "the time interval between B and D should be always greater than or equal to 2.5 seconds and less than 3.5 seconds" can be represented by the instantiated action *IntervalBound(*TRUE*, t, C, D, v, 2.5,* TRUE*, 3.5,* FALSE*)*.

It's noteworthy that when the actions A and B are adjacent, the time interval between A and B denotes the duration of B in fact, according to the time interval semantics we adopt. Formally, as to the definitions of time interval constraints, like $IntervalBound$, when A and B happen adjacently, there is not any action between them, so $Acts =$ FALSE. In such a case, the definition of $IntervalBound$ for A and B is just reduced to the definition of $DurationBound$ for B. It tells that the expression of time intervals between actions is the general case of the time duration of a single action.

The other three actions are variations of $IntervalBound$. The definitions are omitted for the sake of space.

D. Advanced time patterns

Some time concepts like deadline and timeout are usually used in the practical time-sensitive systems. In this section, we provide the corresponding time concepts in TLA$^+$. As a result, both model checking and formal proofs can be made with TLA$^+$ proving rules.

Delay: The request that delays the execution of an action A by val time units is represented in TLA$^+$ by a parameterized action $Delay(A, val, t, vars)$, where t denotes the timer to record this delay ,and $vars$ represents a tuple of variables. The last two parameters are used to compose a correct TLA$^+$ formula. The delay concept is interpreted by a time duration constraint:

$$
\begin{array}{l}
Delay(A, val, t, vars) \triangleq \\
\qquad DurationValue(\text{TRUE}, t, A, vars, val).
\end{array}
$$

Deadine: The deadline of an action A requests that A must terminate before time t. It is represented by $Deadline(A, val, t, vars)$ in TLA$^+$. According to the semantics of deadline, it is interpreted by an upper bound constraint on the duration of A:

$$
\begin{array}{l}
Deadline(A, val, t, vars) \triangleq \\
\qquad DurationUB(\text{TRUE}, t, A, vars, val, \text{TRUE}).
\end{array}
$$

Timeout: The timeout concept is described with $Timeout(A, B, val, t, vars)$ in TLA$^+$. It means that if the action A does not occur in val time units, then execute the action B. It is interpreted by the basic duration constraints

as

$$Timeout(A, B, val, t, vars) \triangleq$$
$$\wedge\ DurationUB(\text{FALSE}, t1, A, vars, val, \text{TRUE})$$
$$\wedge\ DurationValue(\text{TRUE}, t2, B, vars, val, \text{TRUE}).$$

The timeout concept is interpreted by the conjunction of two duration constraints, on A and B, respectively. First, the timeout concept implies that if A occurs, the corresponding timer $t1$ should satisfy $t \leq val$, but the occurrence of A is not requested. This is described by the first conjunct, with the $DurationUB$ on A, as a ***WTR***. On the other hand, the timeout concept also implies that the action B must happen when the timer $t2$ reaches the value val, which is described by the second conjunct, with the action $DurationValue$ as a ***STR***.

IV. The framework of a time specification

With the *RealTimeNew* module, we suggest a framework to write time specifications with TLA$^+$, where the functional and time descriptions are separated to get a clear, well-organized specification, with a good property for specification and verification based on refinement.

A. The description of the framework

Suppose the functional part *Spec* of the time specification is described in a TLA$^+$ module *FuncModule*, and defined with canonical form $Spec \triangleq Init \wedge \Box[Next]_{vars} \wedge L$. The time specification which contains n time constraints can be composed with the following framework:

MODULE RTmodule

EXTENDS $FuncModule, RealTimeNew, Sequences$

VARIABLE $t_1, t_2, \ldots, t_n$

$$BigInit \triangleq Init \wedge now = 0 \bigwedge_{i=1}^{n} t_i = 0$$

$$\begin{aligned} BigNext \triangleq\ & \wedge(Next \vee (\text{UNCHANGED } vars)) \\ & \wedge(NowNext(vars) \vee \text{UNCHANGED } now) \\ & \bigwedge_{i=1}^{n} (\forall \mid \exists j \in SomeSet :)? \\ & DurationBound \mid DurationUB \\ & \mid DurationLB \mid DurationValue \\ & \mid IntevalBound \mid IntervalUB \\ & \mid IntervalLB \mid IntervalValue \\ & \mid Delay \mid Deadline \mid Timeout \end{aligned}$$

$$RTvars \triangleq \langle now, t_1, \ldots, t_n \rangle \circ vars$$

$$RTL \triangleq L \wedge RTFairness(vars)$$

$$RTspec \triangleq BigInit \wedge \Box[BigNext]_{RTvars} \wedge RTL$$

The time specification is defined within a module named ***RTmodule***. The "EXTENDS M" statement adds all the declarations in module M into the module containing this statement. The predefined TLA$^+$ module *Sequences* is included to operate tuples. The n timers are then declared by the VARIABLE statement. The whole time specification is defined by a normal formula ***RTspec***, composed of the initial predicate ***BigInit***, the next state action ***BigNext*** and the fairness constraint ***RTL***.

The initial state of the whole system is composed of the description of the initial state of the functional part, the real time variable *now* and the n timers.

The next state action ***BigNext*** of the whole system is composed of *Next* for the functional part, *NowNext* for time evolving and all the requested time constraints. A time constraint can be an instantiation of any former defined time patterns. Parameters are omitted for the sake of conciseness. The quantified bound may also be included in the time constraints with the form $\forall|\exists\, j \in SomeSet$, where *SomeSet* is the parameter, which is usually a set. The notations are borrowed from regular expressions with the "**?**" symbol denoting option and "|" for choice.

The tuple ***RTvars*** includes the real-time variable *now*, the n timers and all the variables in the functional part. This is defined by the sequence concatenating operator $\circ$ in TLA$^+$.

Finally, the fairness constraint ***RTL*** is the conjunction of ***RTFairness*** on time and the fairness constraint *L* on the functional part.

B. The refinement property of the derived specification

The framework we provide for describing time specifications in TLA$^+$ not only derives a clear and well-organized specification, but also has a refinement property between the time specification ***RTspec*** and its functional part *Spec*. This refinement relationship is denoted by the TLA$^+$ formula

$$RTspec \Rightarrow Spec \tag{1}$$

The proof is easy to get following TLA$^+$ proving rules. Satisfying this refinement relationship is useful for verification. For any property P, if it is satisfied by the behaviors specified in *Spec*, say, $Spec \Rightarrow P$, through the transitivity of implication relation, $RTspec \Rightarrow P$ holds, so it is also satisfied by the behaviors specified in $RTSpec$. In practice, the functional properties can be verified on *Spec*, while the truth are naturally kept on ***RTspec***. Since *Spec* involves less variables, checking properties on it usually requires less space and time. This method therefore provides an efficient way to prove properties, which will be justified by the case study in the next section.

V. Case study

In order to give a fuller account of the time-related representations and to validate the method, we describe its application to a simple answer machine case.

A. The answer machine case

There are a host and several competitors in a competition. The rules are specified as follows.

1) The host and the competitors operate the system through buttons;
2) When the host presses his (or her) button, a new round starts;
3) During a competition round, if some competitor presses his (or her) button within 5 seconds, the corresponding indicator light turns on, and that competitor starts to answer questions; Otherwise, the beeper rings 3 seconds to denote the end of this round;
4) During the question answering phrase, the competitor needs to answer a fixed number of questions. The whole answering phrase should terminate in 20 seconds. After that, the host can push the button to start a new round;
5) Only the first button pressed is valid during a round.
6) To ensure the fairness, when more than one competing buttons are pressed simultaneously, none of them is accepted.

B. The TLA$^+$ specification

The functional specification of the answer machine case in TLA$^+$ is shown in Figure 1. The detailed definitions of the actions like *Compete(i)* and *BeeperRing* are omitted for the space limitation.

```
MODULE answerMachine
EXTENDS Naturals
CONSTANTS
  Competitors, MaxMark, QuestionNum
VARIABLES
  start, button, light, mark, beeper, count, s

Init ≜  ∧ start = FALSE
        ∧ button = [i ∈ Competitors ↦ FALSE]
        ∧ light  = [i ∈ Competitors ↦ FALSE]
        ∧ mark = [i ∈ Competitors ↦ 0]
        ∧ beeper = FALSE
        ∧ count = 0
        ∧ s = "init"
...
Next ≜
  ∨ StartNewRound
  ∨ ∃ i ∈ Competitors : (Compete(i) ∨ Answer(i) ∨ AnswerFinish(i))
  ∨ (BeeperRing ∨ BeeperRingFinish)

vars ≜ ⟨start, button, light, mark, beeper, count, s⟩
L ≜ WF_vars(Next)
Machine ≜ Init ∧ □[Next]_vars ∧ L
```

Figure 1. The functional specification of the answer machine.

This TLA$^+$ specification has three parameters, with *Competitors* denoting the set of competitors, *MaxMark* denoting the maximal mark for a competitor (which is necessary for TLC model checking), and *QuestionNum* denoting the number of questions in the answering phrase.

The next state action *Next* can be a new round starting action operated by the host: *StartNewRound*, the competing and answering action for some competitor i: *Compete(i)*, *Answer(i)* or *AnswerFinish(i)*, or the timeout processing action: *BeeperRing* or *BeeperRingFinish*.

The functional specification is then defined by the formula *machine* in the canonical form, where the liveness constraint is simply to rule out infinite stuttering, obtained by defining L as a weak fairness constraint of the next-state relation.

For the next step, The time specification is given by the formula *RTmachine*, and shown in Figure 2. The first time constraint expresses a timeout constraint on the competing action, which takes a $Timeout$ pattern. The second time constraint denotes a delay of the beeper ringing, which is described by the $Delay$ pattern. The last one expresses the 20 seconds time constraint on the answering phrase, which adopts the basic time interval action to describe it.

```
MODULE RTanswerMachine
EXTENDS answerMachine, RealTimeNew, Sequences
VARIABLE t1, t2, t3

BigInit ≜  ∧ Init
           ∧ now = 0
           ∧ t1 = 0
           ∧ t2 = 0
           ∧ t3 = 0
BigNext ≜  ∧ (Next ∨ (start' = start ∧ UNCHANGED vars)
           ∧ (NowNext(vars) ∨ now' = now)
           ∧ ∀ i ∈ Competitors : Timeout(Compete(i), BeeperRing, 5, t1, vars)
           ∧ Delay(BeeperRingFinish, 3, t2, vars)
           ∧ ∀ i ∈ Competitors :
               IntervalUB(TRUE, t3, Answer(i), AnswerFinish(i), vars, 20, TRUE)
RTvars ≜ ⟨t1, t2, t3, now⟩ ∘ vars
RTL ≜  ∧ RTFairness(vars)
       ∧ L
RTmachine ≜ BigInit ∧ □[BigNext]_RTvars ∧ RTL
```

Figure 2. The time specification of the answer machine.

C. Checking Properties on TLC

The TLC model checker is the main verification tool for TLA$^+$, which can verifies properties over finite instances of TLA$^+$ specifications. To get a finite instance, we consider the case when there are two competitors, the maximal mark $Maxmark$ is 80 and there are two questions to answer in a round ($QuestionNum = 2$). Furthermore, a bounded discrete time model is taken with $now \in 0..50$, instead of the original continuous time model with $now \in Real$. The instantiated specification *RTMachine* is then checked by TLC (10 April 2008 Release), on a computer with an Intel® Core™ 2 duo, T8100 2.10GHz CPU and 3 GB memory.

The first property denotes uniqueness of the accepted answers. This is described by the temporal formula *AtMostOneLightOn*:

$$AtMostOneLightOn \triangleq \Box(\forall i, j \in Competitors : light[i] \wedge light[j] \Rightarrow i = j).$$

Note that *AtMostOneLightOn* is a time-less property. We checked this property on *Machine*, and the TLC model checker verified it in 1.6 seconds. As a result, the property holds for *RTMachine* according to our method. For comparison, we checked the *AtMostOneLightOn* property directly on the *RTMachine* specification. TLC successfully verified it as expected, but with a much larger time 561.1 seconds. The result justified the utility of our separation principle. The specification can be written and verified in a stepwise way. It usually saves verifying efforts since the results hold stepwise owing to the refinement relationship.

We then checked 6 time-related properties on *RTMachine* to further illustrate our time model. They are defined as follows.

$TimeRequest1 \triangleq$
$\Box[\forall i \in Competitors : t1 > 5 \Rightarrow \neg Compete(i))]_{RTvars}$
$TimeRequest2 \triangleq \Box[t1 \leq 5 \Rightarrow \neg BeeperRing]_{RTvars}$
$TimeRequest3 \triangleq$
$\Box[\forall i \in Competitors : AnswerFinish(i) \Rightarrow t3 \leq 20]_{RTvars}$
$TimeRequest4 \triangleq \Box[now \geq t1 \wedge now \geq t2 \wedge now \geq t3]$
$TimeRequest5 \triangleq \Box[t2 > 0 \Rightarrow t1 = 0]$
$TimeRequest6 \triangleq \Box[t3 > 0 \Rightarrow t1 = 0 \wedge t2 = 0]$

TimeRequest1 and *TimeRequest2* together represent the 5 seconds time-out request on the answering phrase. *TimeRequest3* represents that when the answering phrase is finished, the value of timer $t3$ must be less than or equal to 20, for all the competitors.

TimeRequest4, *TimeRequest5* and *TimeRequest6* further explain the time model. *TimeRequest4* specifies that the value of the timer $t1$, $t2$ or $t3$ is never greater than the real time *now*. This is the fundamental property of our time model since a timer counts a certain period of time which is evolving. The last two time properties do not seem so intuitive. *TimeRequest5* declares that when $t2$ is accumulating its value, the value of $t1$ is always zero. By analyzing the actions related to $t1$ and $t2$, we know that $t2 > 0$ only when the action $BeeperRingFinish$ is continuously enabled. On the other hand, when $BeeperRingFinish$ is continuously enabled, both $Compete(i)$ and $BeeperRing$ are disabled, and the value of $t2$ keeps zero in this case. The similar analysis can be taken to validate the *TimeRequest6* property.

We hand the verification procedure to the TLC model checker. All the 6 properties were successfully verified by TLC with a total time of 584.7 seconds. It generates 1,883,403 states, with 144,551 distinct states.

VI. Conclusion

We presented a pattern-based method to express time specifications in the language TLA$^+$. A real-time module *RealTimeNew* is introduced to encapsulate the definitions of commonly used time patterns. Both basic and advanced time patterns are defined to for different applications. A general framework is then presented, which differentiates the temporal characterizations from system functionality with time constraints. The temporal specification is concise and provably as a refinement of its corresponding functional description without time. The framework thus supports the progressive specification and verification. The method is validated by an answer machine case study. The method ameliorates the usability of TLA$^+$ in specifying and verifying time-sensitive systems.

Parameterized specification is permitted in TLA$^+$, while currently the TLC model checker can only process finite and instantiated specification. However, an automatic theorem prover of TLA$^+$ is under development, which will process real numbers and the parameterized specification directly. Our pattern-based method will be adapted to the new verification support of TLA$^+$.

References

[1] Stephan Merz. The specification language TLA$^+$. *Logics of Specification Languages*, pages 401–448, 2008.

[2] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[3] Stephan Merz. TLA$^+$ case study: A resource allocator. Rapport de recherche, LORIA, August 2004.

[4] Gudmund Grov, Greg Michaelson, and Andrew Ireland. Formal verification of concurrent scheduling strategies using TLA. In *ICPADS '07: Proceedings of the 13th International Conference on Parallel and Distributed Systems*, pages 1–6, 2007.

[5] Carlo A. Furia, Dino Mandrioli, Angelo Morzenti, and Matteo Rossi. Modeling time in computing: a taxonomy and a comparative survey. *ACM Computing Surveys*, To appear.

[6] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 2nd edition, 2000.

[7] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems.*, 16(5):1543–1571, September 1994.

[8] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[9] P. Merlin and D. Farber. Recoverability of communication protocols–implications of a theoretical study. *IEEE Transactions on Communications*, 24(9):1036–1043, Sep 1976.

[10] Thorsten Gerdsmeier and Rachel Cardell-Oliver. A method for verifying real-time properties of ada programs. *Engineering of Complex Computer Systems, IEEE International Conference on*, 0:35–47, 2001.

[11] Paul Regnier, George Lima, and Aline Andrade. A TLA+ formal specification and verification of a new real-time communication protocol. *Electronic Notes in Theoretical Computer Science*, 240:221–238, 2009.

[12] Leslie Lamport. *Specifying Systems*. Addison-Wesley, 2002.